A Cache Hierarchy in a Computer System

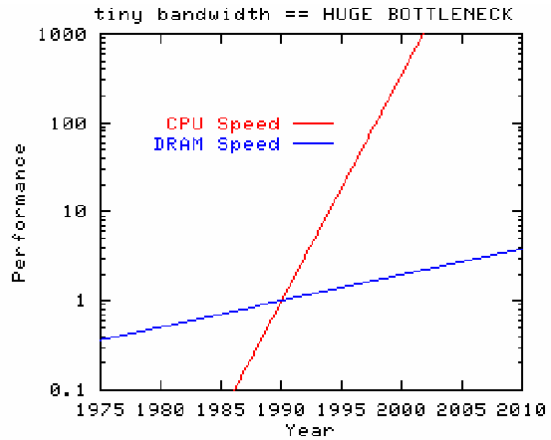
Autor: Paulo J. D. Domingues

A Cache Hierarchy in a Computer System

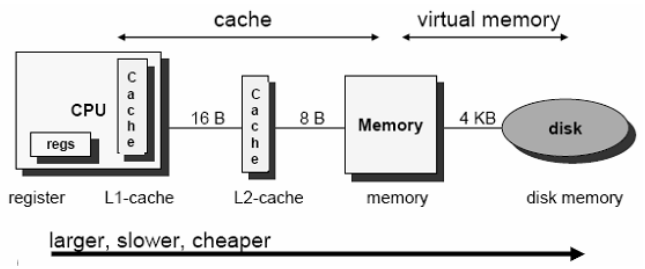
"Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H.Goldstine, and J. Von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument(1946)

Evolution of DRAM and Processor Characteristics



A Memory Hierarchy Example



A Cache Hierarchy in a Computer System

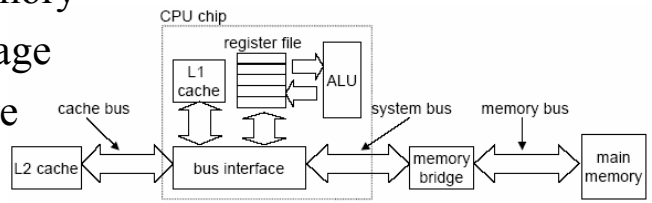
- Why does it works
- Levels of cache
- How does it works
- Comparing & testing

Why Does Cache Works

- ***Exploiting temporal locality***. the same data objects are likely to be reused multiple times..
- ***Exploiting spatial locality***. Blocks usually contain multiple data objects. We can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

Levels of Memory

- Registers
- Cache L1/L2
- Main Memory
- Disk Storage
- Web cache



Typical bus structure for L1 and L2 caches
Bryant and O'Hallaron

Levels of Cache

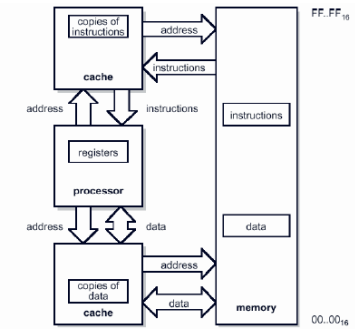
- Primary
  - Always on the CPU;
  - Small (few KB) and very fast;
  - Data and instructions may be separate;
  - P4 has 20 KB
- Secondary
  - Used to be off-chip, now is on chip;
  - Much larger than primary cache;
  - P4 has 512KB on-chip

Characteristics

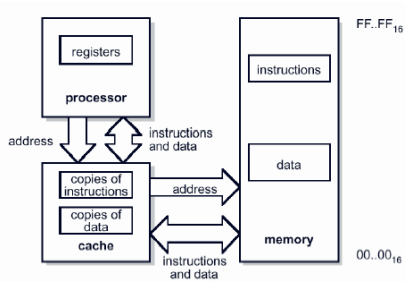
- Cache Size
  - small enough that overall average cost/bit is close to that of main memory alone
  - large enough so that overall average access time is close to that of cache alone
  - large caches tend to be slightly slower than small ones
  - available chip and board area is a limitation
  - studies indicate that L1: 8-64KB, L2: 1-512KB is optimum cache size (for now).
- Speed
  - 7 to 20ns ;

Characteristics

- Separate data and instruction caches



- Unified data and instruction caches

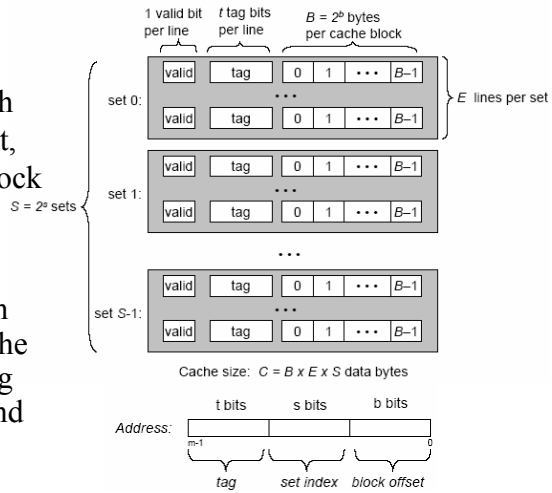


Typical Cache

- It is composed of 2 types of fast memory devices
  - SRAM - hold the actual data, address and status tags in a direct mapped cache
  - TAG RAM - small associative memories
    - help with the accounting duties
    - usually hold at least the address tags for non-direct mapped caches
    - provide fully parallel search capability (sequential search would take too long)

General organization of cache

- A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data.
- The cache organization induces a partition of the *m* address bits into *t* tag bits, *s* set index bits, and *b* block offset bits.



Block Identification

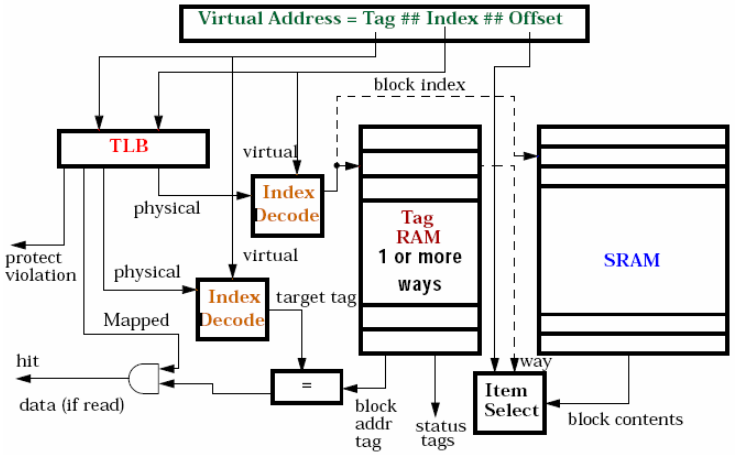
- Components of an address as they relate to the cache:

Tag: Is stored in the cache and used in comparison with CPU address

Index: Used to select the set from the cache

Offset: The first bits of the address give the offset of the byte within a block

General Addressing Model



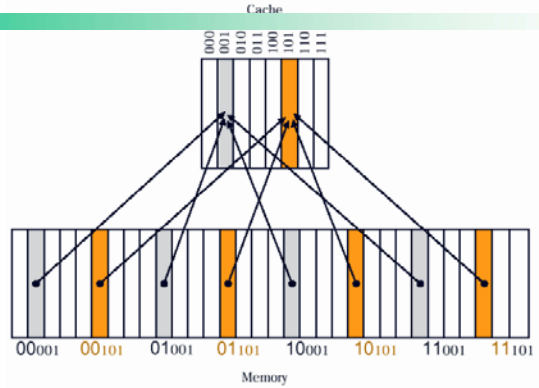
Pentium 4 Cache

- 8-KB Level 1 Data Cache
- Level 1 Execution Trace Cache stores 12-K micro-ops and removes decoder latency from main execution loops
- 512-KB Advanced Transfer Cache (on-die, full-speed Level 2 (L2) cache) with 8-way Associativity

How Cache works

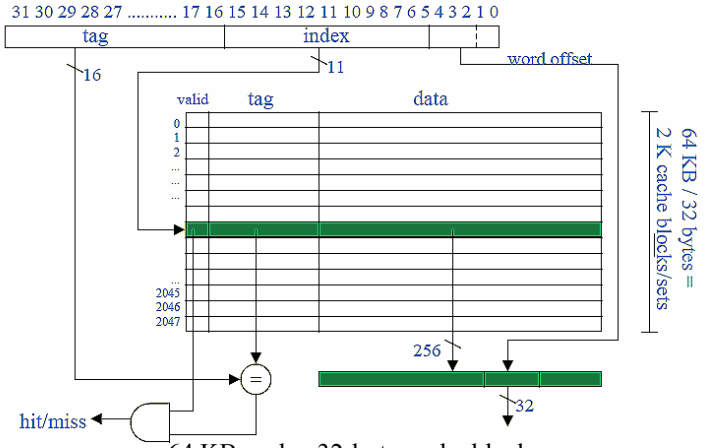
- Cache Mapping
- Replacement of a Cache Block
- Cache Write Policy
- Cache Transfer Technologies
- Improving Cache Performance

Cache Mapping: *Direct mapping*



- Each block of main memory gets a unique mapping;
- If a program happens to repeatedly reference words from two different blocks that map into the same cache line, then the blocks will be continually swapped in the cache and the hit ratio will be low.

Cache Mapping: *Direct mapping*



64 KB cache, 32-byte cache block

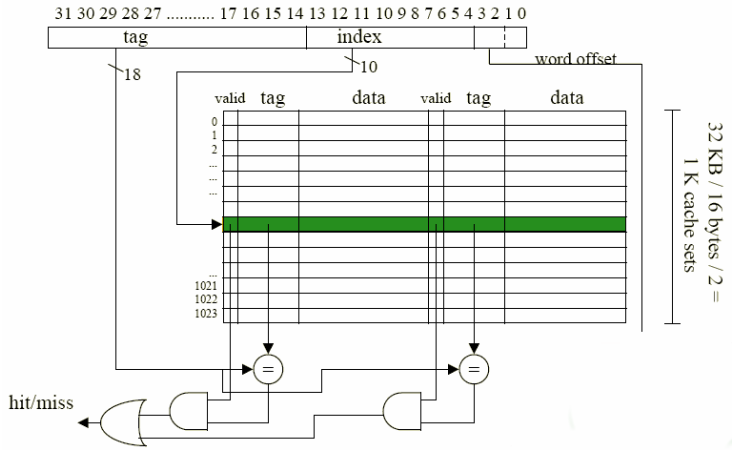
Cache Mapping: *Associative Mapping*

- Allows each memory block to be loaded into any line of the cache
- Tag uniquely identifies a block of main memory
- Cache control logic must simultaneously examine every line's tag for a match
- Requires fully associative memory
  - very complex circuitry
  - complexity increases exponentially with size
  - very expensive

Cache Mapping: *Set Associative Mapping*

- Compromise between direct and associative mappings
- Cache is divided into *v* sets, each of which has *k* lines
  - A given block will map directly to a particular set, but can occupy any line in that set (associative mapping is used within the set)
  - The most common set associative mapping is 2 lines per set, called two-way set associative. *(It significantly improves hit ratio over direct mapping, and the associative hardware is not too expensive.)*

Cache Mapping: *Associative Mapping*



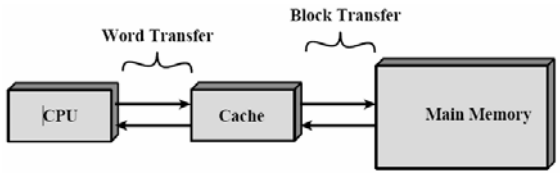
32 KB cache, 2-way set-associative, 16-byte blocks

Replacement of a Cache Block

- Random
  - The simplest method
- Least-Recent Used (LRU)
  - Expensive as cache increase
- First in, first out (FIFO)
  - Simpler than LRU, and almost as good

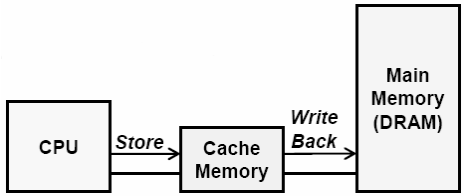
Cache Write Policy

- Write-Back Cache
- Write-Through Cache
  - Buffered Write-Through



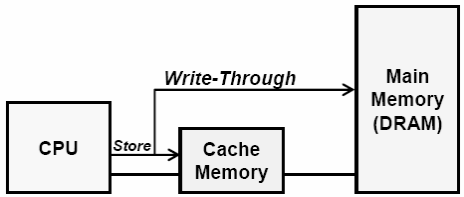
Write-Back Cache

- CPU only updates cache line;
- Modified cache line only written to memory when replaced;
  - Required “dirty-bit” for each cache line
- Memory not always consistent with cache.



Write-Through Cache

- CPU updates both cache and memory;
- Memory is always consistent with cache;
- Usually is combined with *write buffers,* meaning that CPU doesn't have to wait for memory to complete write.



Cache Transfer Technologies

- Cache Bursting
  - Once the transfers are done in sequence there's no need to specify a different address after the first one.
- Asynchronous Cache
  - Transfers are not tied to the system clock. Is very slow for higher clock cycles.
- Synchronous Burst Cache
  - Each tick of the system clock, a transfer can be done to or from the cache.
- Pipeline Burst (PLB) Cache
  - The data transfers that occur in a "burst" are done partially at the same time. The second transfer begins before the first transfer is done.

Improving Cache Performance

- Reducing Miss Penalty
  - Additional time required because of a miss
- Reducing Miss Rate
  - The fraction of memory references during the execution of a program
- Reducing Miss Penalty or Rate via Parallelism
- Reducing Hit Time
  - Time taken to deliver a word in the cache to the CPU

Reducing Miss Penalty

- Multilevels Caches
  - Small and faster L1; larger L2
- Critical Word First and Early Restart
  - As soon as first word of block arrives it's delivered to CPU
- Giving Priority to Read Miss over Writes
  - The dirty block is written into a buffer
- Merging Write Buffer
  - Combine data without having to write immediately to memory
- Victim Caches
  - Small, fully associative cache between L1 and L2

Reducing Miss Rate

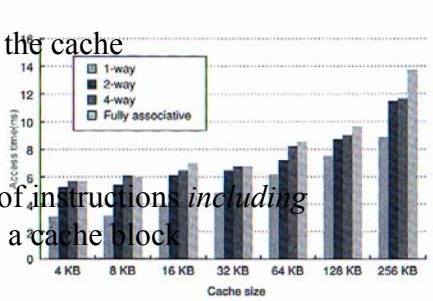
- Larger Block Size
  - It is likely that the next word will be after the last one
- Larger Caches
  - Increases cost and hit time
- Higher Associativity
  - An eight-way set associative cache is as effective as a full associative cache
  - The more a cache is associative the higher the hit time.
- Way Prediction and PseudoAssociative Caches
  - Extra bits are needed to early set the multiplexor to select the desired block
- Compiler Optimizations
  - Merging Arrays, Loop Interchange, Loop Fusion, Blocking

Reducing Miss Penalty or Rate via Parallelism

- Nonblocking Caches to Reduce stalls on cache Misses
  - The cache can continue to supply cache hits during a miss.
- H/W Prefetching of Instructions and Data
  - The processor on a cache misses fetches besides the requested blocks, the next one.
- Compiler-Controlled Prefetching
  - Prefetching, performed by the compiler

Reducing Hit Time

- Small and Simple Caches
  - small hardware is faster
- Avoiding Address Translation during Indexing of the Cache
  - Using virtual addresses for the cache
- Pipelined Cache Access
- Trace Caches
  - Finds a dynamic sequence of instructions *including taken branches* to load into a cache block



Cache Optimization Summary

Technique	Miss Rate	Miss Pen.	Hit time	Hardware Complexity
Larger Block Size	+	-		0
Higher Associativity	+		-	1
Victim Caches	+			2
Pseudo-associative	+			2
Hardware Prefetching	+			2
Compiler-controlled Pre	+			3
Compiler Techniques	+			0
Giving Read Misses Priority		+		1
Subblock Placement		+		1
Early Restart/Crit Wd First		+		2
Nonblocking Caches		+		3
Second-Level Caches		+		2
Small and Simple Caches	-		+	0
Avoiding Address Trans.			+	2
Pipelining Writes			+	1

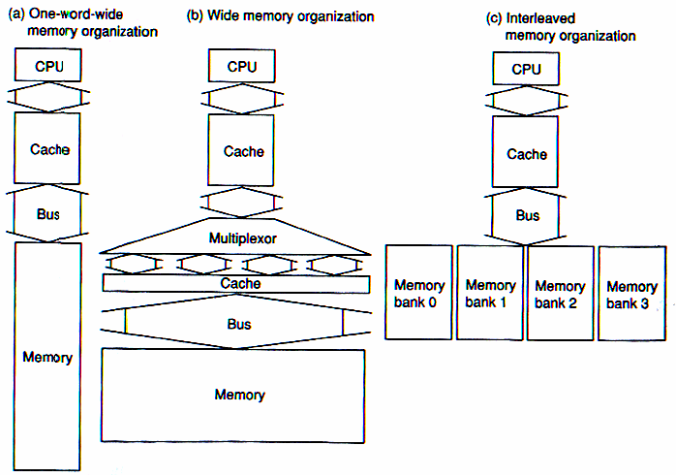
Main Memory

- ***Latency***
  - Affects primarily cache, in the miss penalty
- ***Bandwidth***
  - Has more effect on I/O and multiprocessors

Improve Memory Bandwidth

- Wider Main Memory
  - Twice the width, half the memory accesses
- Simple Interleaved Memory
  - Several chips in a memory system organized in *banks* (each bank can be one word width)
- Independent Memory Banks
  - Multiple independent accesses where multiple memory controllers allow banks to operate independently

Memory Bandwidth



Cache Coherence Problem

- Multiple copies of the same data in different caches;
- Data in the caches is modified locally;
- Cache and memory can be incoherent;
- Data may become inconsistent between caches on multiprocessor systems.

Cache Coherence Problem - *Solutions*

- Declare shared data to be non-cacheable (by software)
- Cache on read, but not cache write cycles;
- Use bus snooping to maintain coherence on write back to main memory.
- Or Directory based: Cache lines maintain an extra 2 bits per processor to preserve clean/dirty status bits.

Exercices- Is the following code Cache-friendly?

```
#define N 1000

typedef struct {
  int vel[3];
  int acc[3];
} point;

point p[N];
```

An array of structs.

Exercices- Is the following code Cache-friendly?

```
void clear1(point *p, int n)
 {
 int i, j;

 for (i = 0; i < n; i++)
 {
   for (j = 0; j < 3; j++)
      p[i].vel[j] = 0;
   for (j = 0; j < 3; j++)
      p[i].acc[j] = 0;
 }
}
```


Exercices- Is the following code Cache-friendly?

```
void clear2(point *p, int n)
{
  int i, j;

  for (i = 0; i < n; i++)  {
    for (j = 0; j < 3; j++) {
      p[i].vel[j] = 0;
      p[i].acc[j] = 0;
    }
  }
}
```

Exercices- Is the following code Cache-friendly?

```
void clear3(point *p, int n)
{
  int i, j;

  for (j = 0; j < 3; j++) {
    for (i = 0; i < n; i++)
      p[i].vel[j] = 0;
    for (i = 0; i < n; i++)
      p[i].acc[j] = 0;
  }
}
```